

Towards MKM in the Large: Modular Representation and Scalable Software Architecture ^{*}

Michael Kohlhase, Florian Rabe, Vyacheslav Zholudev

Computer Science, Jacobs University Bremen
{m.kohlhase,f.rabe,v.zholudev}@jacobs-university.de

Abstract. MKM has been defined as the quest for technologies to manage mathematical knowledge. MKM “in the small” is well-studied, so the real problem is to scale up to large, highly interconnected corpora: “MKM in the large”. We contend that advances in two areas are needed to reach this goal. We need representation languages that support incremental processing of all primitive MKM operations, and we need software architectures and implementations that implement these operations scalably on large knowledge bases.

We present instances of both in this paper: the MMT framework for modular theory-graphs that integrates meta-logical foundations, which forms the base of the next OMDoc version; and TNTBase, a versioned storage system for XML-based document formats. TNTBase becomes an MMT database by instantiating it with special MKM operations for MMT.

1 Introduction

[12] defines the objective of MKM to be *to develop new and better ways of managing mathematical knowledge using sophisticated software tools* and later states the “Grand Challenge of MKM” as a *universal digital mathematics library (UDML)*, which is indeed a grand challenge, as it envisions that the UDML *would continuously grow and in time would contain essentially all mathematical knowledge*, which is estimated to be well in excess of 10^7 published pages.¹ All current efforts towards comprehensive machine-organizable libraries of mathematics are at least three orders of magnitude smaller than the UDML envisioned by Farmer in 2004: Formal libraries like those of Mizar ([33], Isabelle ([26]) or PVS ([25]) have ca. $10^{4..x}$ statements (definitions and theorems). Even the semi-formal, commercial Wolfram MathWorld which hails itself *the world’s most extensive mathematics resource* only has $10^{4.1}$ entries. There is anecdotal evidence that already at this size, management procedures are struggling.

To meet the MKM Grand Challenge will have to develop fundamentally more scalable ways of dealing with mathematical knowledge, especially since [12] goes on to postulate that the UDML *would also be continuously reorganized and consolidated as new connections and discoveries were made*. Clearly this can only be achieved algorithmically; experience with the libraries cited above already show that manual MKM

^{*} The final publication of this paper is available at www.springerlink.com.

¹ For instance, Zentralblatt Math contains 2.4 million abstracts of articles from mathematical journals in the last 100 years.

does not scale sufficiently. Most of the work in the MKM community has concentrated on what we could call “MKM in the small”, i.e. dealing with aspects of MKM that do not explicitly address issues of very large knowledge collections; these we call “MKM in the large”.

In this paper we contribute to the MKM Grand Challenge of doing formal “MKM in the large” by analyzing scalability challenges inherent in MKM and propose steps towards solutions based on our MMT format, which is the basis for the next version of OMDOC. We justify our conclusions and recommendations for scalability techniques with concrete case studies we have undertaken in the last years. Section 2 tackles scalability issues pertaining to the representation languages used in the formalization of mathematical knowledge. Section 3 discusses how the modularity features of MMT can be realized scalably by realizing basic MKM functionality like validation, querying, and presentation incrementally and carefully evaluating the on-the-fly computation (and caching) of induced representations. These considerations, which are mainly concerned with efficient computation “in memory” are complemented with a discussion of mass storage, caching, and indexing in Section 4, which addresses scalability issues in large collections of mathematical knowledge. Section 5 concludes the paper and addresses avenues of further research.

2 A Scalable Representation Language

Our representation language MMT was introduced in [29]. It arises from three central design goals. Firstly, it should provide an expressive but simple **module system** as modularity is a necessary requirement for scalability. As usual in language design, the goals of simplicity and expressivity form a difficult trade-off that must be solved by identifying the right primitive module constructs. Secondly, scalability across semantic domains requires **foundation-independence** in the sense that MMT does not commit to any particular foundation (such as Zermelo-Fraenkel set theory or Church’s higher-order logic). Providing a good trade-off between this level of generality and the ability to give a rigorous semantics is a unique feature of MMT. Finally, scalability across implementation domains requires **standards-compliance**, and while using XML and OPENMATH is essentially orthogonal to the language design, the use of URIs as identifiers is not as it imposes subtle constraints that can be very hard to meet a posteriori.

MMT represents logical knowledge on three levels: On the **module level**, MMT builds on modular algebraic specification languages for logical knowledge such as OBJ [14], ASL [32], development graphs [1], and CASL [7]. In particular, MMT uses theories and theory morphism as the primitive modular concepts. Contrary to them, MMT only imposes very lightweight assumptions on the underlying language. This leads to a very simple generic module system that subsumes almost all aspects of the syntax and semantics of specific module systems such as PVS [25], Isabelle [26], or Coq [3].

On the **symbol level**, MMT is a simple declarative language that uses named symbol declarations where symbols may or may not have a type or a definiens. By experimental evidence, this subsumes virtually all declarative languages. In particular, MMT uses the Curry-Howard correspondence [8,17] to represent axioms and theorem as con-

stants, and proofs as terms. Sets of symbol declarations yield theories and correspond to OPENMATH content dictionaries.

On the **object level**, MMT uses the formal grammar of OPENMATH [6] to represent mathematical objects without committing to a specific formal foundation. The semantics of objects is given proof theoretically using judgments for typing and equality between objects. MMT is parametric in these judgments, and the precise choice is relegated to a **foundation**.

2.1 Module System

Sophisticated mathematical reasoning usually involves several related but different mathematical contexts, and it is desirable to exploit these relationships by moving theorems between contexts. It is well-known that modular design can reduce space to an extent that is exponential in the depth of the reuse relation between the modules, and this applies in particular to the large theory hierarchies employed in mathematics and computer science.

The first applications of this technique in mathematics are found in the works by Bourbaki ([4,5]), which tried to prove every theorem in the context with the smallest possible set of axioms. MMT follows the “little theories approach” proposed in [11], in which separate contexts are represented by separate **theories**, and structural relationships between contexts are represented as **theory morphisms**, which serve as conduits for passing information (e.g., definitions and theorems) between theories (see [10]). This yields **theory graphs** where the nodes are theories and the paths are theory morphisms.

Example 1 (Algebra). For example, consider the theory graph in Fig. 1 for a portion of algebra, which was formalized in MMT in [9]. It defines the theory of magmas (A magma has a binary operation without axioms.) and extends it successively to monoids, groups, and commutative groups. Then the theory of rings is formed by importing from both CGroup (for the additive operation) and Monoid (for the multiplicative operation).

A crucial property here is that the imports are named, e.g., Monoid imports from Magma via an import named `mag`. While redundant in some cases, it is essential in Ring where we have to distinguish two theory morphisms from Monoid to Ring: The composition `add/grp/mon` for the additive monoid and `mult` for the multiplicative monoid.

The import names are used to form qualified names for the imported symbols. For example, if `*` is the name of the binary operation in Magma, then `add/grp/mon/mag/*` denotes addition and `mult/mag/*` multiplication in Ring. Of course, MMT supports the use of abbreviations instead of qualified names, but it is a crucial prerequisite for scalability to make qualified names the default: Without named imports, every time we add a new name in Magma (e.g. for an abbreviation or a theorem), we would have to add corresponding renamings in Ring to avoid name clashes.

Another reason to use named imports is that we can use them to instantiate imports with theory morphisms. In our example, distributivity is stated separately as a theory that imports two magmas. Let us assume, the left distributivity axiom is stated as

$$\forall x, y, z. x \text{ mag1}/* (y \text{ mag2}/* z) = (x \text{ mag1}/* y) \text{ mag2}/* (x \text{ mag1}/* z)$$

Then the import *dist* from *Distrib* to *Ring* will carry the instantiations $\text{mag1} \mapsto \text{mult}/\text{mag}$ and $\text{mag2} \mapsto \text{add}/\text{grp}/\text{mon}/\text{mag}$.

In other module systems such as SML, such instantiations are called (asymmetric) sharing declarations. In terms of theory morphism, their semantics is a commutative diagram, i.e., an equality between two morphisms such as $\text{dist}/\text{mag1} = \text{mult}/\text{mag} : \text{Magma} \rightarrow \text{Ring}$. This provides MMT users and systems with a module level invariant for the efficient structuring of large theory graphs.

Besides imports, which induce theory morphisms into the containing theory, there is a second kind of edge in the theory graph: Views are explicit theory morphisms that represent translations between two theories. For example, the node on the right side of the graph represents a theory for the integers, say declaring the constants 0, +, −, 1, and ·. The fact that the integers are a commutative group is represented by the view *v1*: If we assume that *Monoid* declares a constant *e* for the unit element and *Group* a constant *inv* for the inverse element, then *v1* carries the instantiations $\text{grp}/\text{mon}/\text{mag}/\cdot \mapsto +$, $\text{grp}/\text{mon}/e \mapsto 1$, and $\text{grp}/\text{inv} \mapsto -$. Furthermore, every axiom declared or imported in *CGroup* is mapped to a proof of the corresponding property of the integers.

The view *v2* extends *v1* with corresponding instantiations for multiplication. MMT permits modular views as well: When defining *v2*, we can import all instantiations of *v1* using $\text{add} \mapsto \text{v1}$. As above, the semantics of such an instantiation is a commutative diagram, namely $\text{v2} \circ \text{add} = \text{v1}$ as intended.

The major advantage of modular design is that every declaration — abbreviations, theorems, notations etc. — effects **induced declarations** in the importing theories. A disadvantage is that declarations may not always be located easily, e.g., the addition in a ring is declared in a theory that is four imports away. MMT finds a compromise here: Through qualified names, all induced declarations are addressable and locatable. The process of removing the modularity by adding all these induced declarations to all theories is called **flattening**.

Case Study 1: The formalization in [9] uses the Twelf module system ([31]), which is a special case of MMT. Twelf automatically computes the flattened theory graph. The modular theory graph including all axioms and proofs can be written in 180 lines of Twelf code. The flattened graph is computed in less than half a second and requires more than 1800 lines.

The same case study defines two theories for lattices, one based on orderings and one based on algebra, and gives mutually inverse views to prove the equivalence of the two theories. Both definitions are modular: Algebraic lattices arise by importing twice from the theory of semi-lattices; order-based lattices arise by importing the infimum operation twice, once for the ordering and once for its dual. Consequently, the views can be given modularly as well, which is particularly important because views must map axioms to expensive-to-find proofs. These additional declarations take 310 lines of Twelf in modular and 3500 lines in flattened form.

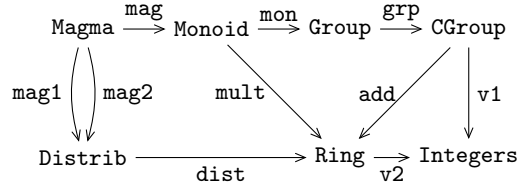


Fig. 1. Algebraic Hierarchy

These numbers already show the value of modularity in representation already in very small formalizations. If this is lacking, later steps will face severe scalability problems from blow-up in representation. Here, the named imports of MMT were the crucial innovation to strengthen modularity.

2.2 Foundation-Independence

Mathematical knowledge is described using very different foundations, and the most common foundations can be grouped into set theory and type theory. Within each group there are numerous variants, e.g., Zermelo-Fraenkel or Gödel-Bernays set theory, or set theories with or without the axiom of choice. Therefore, a single representation language can only be adequate if it is foundation-independent.

OPENMATH and OMDOC achieve this by concentrating on structural issues and leaving lexical ones to an external definition mechanism like content dictionaries or theories. In particular, this allows us to operate without choosing a particular foundational logical system, as we can just supply content dictionaries for the symbols in the particular logic. Thus, logics and in the same way foundations become theories, and we speak of the **logics-as-theories** approach.

But conceptually, it is helpful to distinguish levels here. To state a property in the theory CGroup like commutativity of the operation $\circ := \text{grp/mon/mag}/*$ as $\forall a, b. a \circ b = b \circ a$, we use symbols \forall and $=$ from first-order logic together with \circ from CGroup. Even though it is structurally possible to build algebraic theories by simply importing first-order logic, this would fail to describe the meta-relationship between the theories. But this relation is crucial, e.g., when interpreting CGroup in the integers, the symbols of the meta-language are not interpreted because a fixed interpretation is given in the context.

To understand this example better, we use the M/T notation for meta-languages. M/T refers to working in the object language T , which is defined within the meta-language M . For example, most of mathematics is carried out in FOL/ZF , i.e., first-order logic is the meta-language, in which set theory is defined. FOL itself might be defined in a logical framework such as LF , and within ZF , we can define the language of natural numbers, which yields $LF/FOL/ZF/Nat$. For algebra, we obtain, e.g., $FOL/Magma$. MMT makes this meta-relation explicit: Every theory T may point to another theory M as its meta-theory. We can write this as $MMT/(M/T)$.

In Fig. 2, the algebra example is extended by adding meta-theories. The theory FOL for first-order logic is the meta-theory for all algebraic theories, and the theory LF for the logical framework LF is the meta-theory of FOL and of the theory HOL for higher-order logic.

Now the crucial advantage of the logics-as-theories approach is that on all three levels the same module system can be used: For example, the views m and m' indicate possible translations on the levels

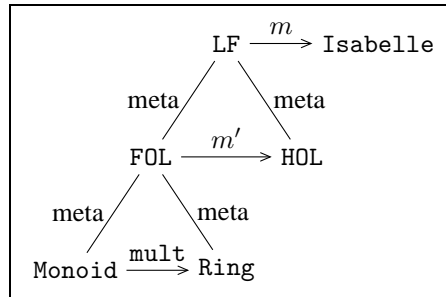


Fig. 2. Meta-Theories

of logical frameworks and logics, respectively. Similarly, logics and foundations can be built modularly. Thus, we can use imports to represent inheritance at the level of logical foundations and views to represent formal translations between them. Just like in the little theories approach, we can prove meta-logical results in the simplest foundation that is expressive enough and then use views to move results between foundations.

Example 2 (Little Logics and Little Foundations). In [15], we formalize the syntax, proof theory, and model theory and prove the soundness of first-order logic in MMT. Using the module system, we can treat all connectives and quantifiers separately. Thus, we can reuse these fragments to define other logics, and in [18] we do that, e.g., for sorted first-order logic and modal logic.

For the definition of the model theory, we need to formalize set theory in MMT, which is a significant investment, and even then doing proofs in set theory — as needed for the soundness proof — is tedious. Therefore, in [16], we develop the set theoretical foundation itself modularly. We define a typed higher-order logic HOL first, which is expressive enough for many applications such as the above soundness proof. Then a view from HOL to ZF proves that ZF is a refinement of HOL and completes the proof of the soundness of FOL relative to models defined in ZF.

Case Study 2: Ex. 2 already showed that it is feasible to represent foundations and relations between foundations in MMT. Being able to this is a qualitative aspect of cross-domain scalability. In another case study, we represented *LF/Isabelle* and *LF/Isabelle/HOL* ([26,23]) as well as a translation from them into *LF/FOL/ZFC* (see [18]).

To our knowledge, MMT is the only declarative formalism in which comparable foundation or logic translations have been conducted. In Hets ([21]) a number of logic translations are implemented in Haskell. Twelf and Delphin provide logic and functional programming languages, respectively, on top of LF ([27,28]), which have been used to formalize the HOL-Nuprl translation ([22]).

2.3 Symbol Identifiers “in the Large”

In mathematical languages, we need to be able to refer to (i.e., identify) content objects in order to state the semantic relations. It was a somewhat surprising realization in the design of MMT that to understand the symbol identifiers is almost as difficult as to understand the whole module system. Theories are containers for symbol declarations, and relations between theories define the available symbols in any given theory. Since every available symbol should have a canonical identifier, the syntax of identifiers is inherently connected to the possible relations between theories.

In principle, there are two ways to identify content object: **by location** (relative to a particular document or file) and **by context** (relative to a mathematical theory). The first one essentially makes use of the organizational structure of files and file systems, and the second makes use of mathematical structuring principles supplied by the representation format.

As a general rule, it is preferable to use identification by context as the distribution of knowledge over file systems is usually a secondary consideration. Then the mapping

between theory identifiers and physical theory locations can be deferred to an extralinguistic catalog. Resource identification by context should still be compatible with the URI-based approach that mediates most resource transport over the internet. This is common practice in scalable programming languages such as Java where package identifiers are URIs and classes are located using the `classpath`.

For logical and mathematical knowledge, the OPENMATH 2 standard ([6]) and the current OMDOC version 1.2 define URIs for symbols. A symbol is identified by the symbol name and content dictionary, which in turn is identified by the CD name and the CD base, i.e., the URI where the CD is located. From these constituents, symbol URIs are formed using URI fragments (the part after the # delimiter). However, OPENMATH imposes a one-CD-one-file restriction, which is too restrictive in general. While OMDOC1.2 permits multiple theories per file, it requires file-unique identifiers for all symbols. In both cases, the use of URI fragments, which are resolved only on the client, forces clients to retrieve the complete file even if only a single symbol is needed.

Furthermore, many module systems have features that impede or complicate the formation of canonical symbol URIs. Such features include unnamed imports, unnamed axioms, overloading, opening of modules, or shadowing of symbol names. Typically, this leads to a non-trivial correspondence between user-visible and application-internal identifiers. But this impedes or complicates cross-application scalability where all applications (ranging from, e.g., a Javascript GUI to a database backend) must understand the same identifiers.

MMT avoids the above pitfalls and introduces a simple yet expressive web-scalable syntax for symbol identifiers. An MMT-URI is of the form *doc?mod?sym* where

- *doc* is a URI without query or fragment, e.g., <http://cds.omdoc.org/math/algebra/algebra1.omdoc> which identifies (but not necessarily locates) an MMT document,
- *mod* is a /-separated sequence of local names that gives the path to a nested theory in the above document, e.g., `Ring`,
- *sym* is a /-separated sequence *imp₁/.../imp_n/con* of local names such that *imp_i* is an import and *con* a symbol name, e.g., `mult/mon/*`,
- a local name is of the form *pchar*⁺ where *pchar* is defined as in RFC 3986 [2], which — possibly via %-encoding — permits almost all Unicode characters.

In our running example, the canonical URI of multiplication in a ring is <http://cds.omdoc.org/math/algebra/algebra1.omdoc#Ring/mult>. Note that the use of two ? characters in a URI is unusual outside of MMT, but legal w.r.t. RFC 3986. Of course, MMT also defines relative URIs that are resolved against the URI of the containing declaration. The most important case is when *doc* is empty. Then the resolution proceeds as in RFC 3986, e.g., *?mod'?sym'* resolves to *doc?mod'?sym'* relative to *doc?mod?sym* (Note that this differs from RFC 2396.). MMT defines some additional cases that are needed in mathematical practice and go beyond the expressivity of relative URIs: Relative to *doc?mod?sym*, the resolution of *??sym'* and *?/mod'?sym'* yields *doc?mod?sym'* and *doc?mod/mod'?sym'*, respectively.

Case Study 3: URIs are the main data structure needed for cross-application scalability, and our experience shows that they must be implemented by almost every peripheral system, even those that do not implement MMT itself. Already at this point, we had to implement them in SML ([31]), Javascript ([13]), XQuery ([35]), Haskell (for Hets,

[21]), and Bean Shell (for a jEdit plugin) — in addition to the Scala-based reference API (Sect. 3).

This was only possible because MMT-URIs constitute a well-balanced trade-off between mathematical rigor, feasibility, and URI-compatibility: In particular, due to the use of the two separators `/` and `?` (rather than only one), they can be parsed locally, i.e., without access to or understanding of the surrounding MMT document. And they can be dereferenced using standard URI libraries and URI-URL translations. At the same time, they provide canonical names for all symbols that are in scope, including those that are only available through imports.

3 A Scalable Implementation

As the implementation language for the MMT reference API, we pick Scala, a programming language designed to be *scalable* ([24]). Being functional, Scala permits elegant code, and based on and fully compatible with Java, it offers cross-application and web-level scalability.

The MMT API implements the syntax and semantics of MMT. It compiles to a 1 MB Java archive file that can be readily integrated into applications. Library and documentation can be obtained from [30]. Two technical aspects are especially notable from the point of view of scalability. Firstly, all MMT functionality is exposed to non-Java applications via a scriptable shell and via an HTTP servlet. Secondly, the API maintains an abstraction layer that separates the backends that physically store MMT documents (URLs) from the document identifiers (URIs). Thus, it is configurable which MMT documents are located, e.g., in a remote database or on the local file system. In the following section we describe some of the advanced features.

3.1 Validation

Validation describes the process of checking MMT theory graphs against the MMT grammar and type system. MMT validation is done in three increasingly strict **stages**.

The first stage is XML validation against a context-free RelaxNG grammar. As this only catches errors in the surface syntax, MMT documents are validated **structurally** in a second stage. Structural validity guarantees that all declarations have unique URIs and that all references to symbols, theories, etc. can be resolved. This is still too lax for mathematics as it lacks type-checking. But it is exactly the right middle ground between the weak validation against a context-free grammar and the expensive and complex validation against a specific type system: On the one hand, it is efficient and foundation-independent, and on the other hand, it provides an invariant that is sufficient for many MKM services such as storage, navigation, or search.

Type-checking is foundation-specific, therefore each foundation must provide an MMT plugin that implements the typing and equality judgments. More precisely, the plugin must provide function that (semi-)decide for two given terms A and B over a theory T , the judgments $\vdash_T A = B$ and $\vdash_T A : B$. Given such a plugin, a third validation stage can refine structural validity by additionally validating well-typedness of all

declarations. For scalability, it is important that (i) these plugins are stateless as the theory graph is maintained by MMT, and that the (ii) modular structure is transparent to the plugin. Thus plugin developers only need to provide the core algorithms for the specific type system, and all MKM issues can be relegated to dedicated implementations.

Context-free validation is well-understood. Moreover, MMT is designed such that foundation-specific validation is obtained from structural validation by using the same inference system with some typing and equality constraints added. This leaves structural validation as the central issue for scalability.

Case Study 4: We have implemented structural validation by decomposing an MMT theory graph into a list of atomic declarations. For example, the declaration $T = \{s_1 : \tau_1, s_2 : \tau_2\}$ of a theory T with two typed symbols yields the atomic declarations $T = \{\}, T?s_1 : \tau$, and $T?s_2 : \tau_2$. This “unnesting” of declarations is a special property of the MMT type system that is not usually found in other systems. It is possible because every declaration has a canonical URI and can therefore be taken out of its context.

This is important for scalability as it permits **incremental** processing. In particular, large MMT documents can be processed as streams of atomic declarations. Furthermore, the semantics of MMT guarantees that the processing order of these streams never matters if the (easily-inferable) dependencies between declarations are respected. This would even permit parallel processing, another prerequisite for scalability.

3.2 Querying

Once a theory graph has been read, MMT provides two ways how to access it: MMT-URI dereferencing and querying with respect to a simple ontology.

Firstly, a theory graph always has two forms: the modular form where all nodes are partial theories whose declarations are computed using imports, and the flattened form where all imports are replaced with translated copies of the imported declarations. Many implementations of module systems, e.g., Isabelle’s locales, automatically compute the flat form and do not maintain the modular form. This can be a threat to scalability as it can induce combinatorial explosion.

MMT maintains only the modular form. However, as every declaration present in the flat form has a canonical URI, the access to the flat form is possible via MMT-URI dereferencing: Dereferencing computes (and caches) the induced declarations present in the flat form. Thus, applications can ignore the modular structure and interact with a modular theory graph as if it were flattened, but the exponentially expensive flattening is performed transparently and incrementally.

Secondly, the API computes the ABox of a theory graph with respect to the MMT ontology. It has MMT-URIs as individuals and 10 types like `theory` or `symbol` as unary predicates. 11 binary predicates represent relations between individuals such as `HasDomain` relating an import to a theory or `HasOccurrenceOfInType` relating two symbols. These relations are structurally complete: The structure of a theory graph can be recovered from the ABox. The computation time is negligible as it is a byproduct of validation anyway.

The API includes a simple query language for this ontology. It can compute all individuals in the theory graph that are in a certain relation to a given individual. The possible queries include composition, union, transitive closure, and inverse of relations.

The ABox can also be regarded as the result of **compiling** an MMT theory graph. Many operations on theory graphs only require the ABox: for example the computation of the forward or backward dependency cone of a declaration which are needed to generate self-contained documents and in change management, respectively. This is important for cross-application scalability because applications can parse the ABox very easily. Moreover, we obtain a notion of **separate compilation**: ABox-generation only requires structural validity, and the latter can be implemented if only the ABoxes of the referenced files are known.

Case Study 5: Since all MMT knowledge items have a globally unique MMT-URI, being able to dereference them is sufficient to obtain complete information about a theory graph. We have implemented a web servlet for MMT that can act as a local proxy for MMT-URIs and as a URI catalog that maps MMT-URIs into (local or remote) URLs. The former means that all MMT-URIs are resolved locally if possible. The latter means that the MMT-URI of a module can be independent from its physical location. The same servlet can be run remotely, e.g., on the same machine as a mathematical database and configured to retrieve files directly from there or from other servers.

Thus systems can access all their input documents by URI via a local service, which makes all storage issues transparent. (Using presentation, see below, these can even be presented in the system's native syntax.) This solves a central problem in current implementations of formal systems: the restriction to in-memory knowledge. Besides the advantages of distributed storage and caching, a simple example application is that imported theories are automatically included when remote documents are retrieved in order to avoid successive lookups.

3.3 Presentation

MMT comes with a declarative language for notations similar to [19] that can be used to transform MMT theory graphs into arbitrary output formats. Notations are declared by giving parameters such as fixity and input/output precedence, and snippets for separators and brackets. Notations are not only used for mathematical objects but also for all MMT expressions, e.g. theory declarations and theory graphs.

Two aspects are particularly important for scalability. Firstly, sets of notations (called *styles*) behave like theories, which are sets of symbols. In particular, styles and notations have MMT-URIs (and are part of the MMT ontology), and the MMT module system can be used for inheritance between styles.

Secondly, every MMT expression has a URI E , for declarations this is trivial, for most mathematical objects it is the URI of the head symbol. Correspondingly, every notation must give an MMT-URI N , and the notation is applicable to an expression if N is a prefix of E . More specific notations can inherit from more general ones, e.g., the brackets and separators are usually given only once in the most general notation. This simplifies the authoring and maintenance of styles for large theory graphs significantly.

Case Study 6: In order to present MMT content as, e.g., HTML with embedded presentation MATHML, we need a style with only the 20 generic notations given in <http://alpha.tntbase.mathweb.org/repos/cds/omdoc/mathml.omdoc>.

For example, the notation declaration on the right applies to all constants whose `cdbase` starts with <http://cds.omdoc.org/> and renders OMS elements as `mo` elements. The latter has an `xref` attribute that links to the parallel markup (which is included by notations at higher levels). The content of the

```
<notation for="http://cds.omdoc.org/"
  role="constant">
  <element name="mo">
    <attribute name="xref">
      <text value="#" /></id/>
    </attribute>
    <hole><component index="2" /></hole>
  </element>
</notation>
```

`mo` elements is a “hole” that is by default filled with the second component, for constants that is the name (0 and 1 are `cdbase` and `cd.`).

This scales well because we can give notations for specific theories, e.g., by saying that *Magma*?* is associative infix and optionally giving a different operator symbol than *. We can also add other output formats easily: Our implementation (see [18]) extends the above notation with a `jobad:href` attribute containing the MMT-URI — this URI is picked up by our JOBAD Javascript ([13]) for hyperlinking.

4 A Scalable Database

The TNTBase system [34] is a versioned XML-database with a client-server architecture. It integrates Berkeley DB XML into a Subversion server. DB XML stores HEAD revisions of XML files; non-XML content like PDF, images or \LaTeX source files, differences between revisions, directory entry lists and other repository information are retained in a usual SVN back-end storage (Berkeley DB in our case). Keeping XML documents in DB XML allows accessing files not only via any SVN client but also through the DB XML API that supports efficient querying of XML content via XQuery and (versioned) modification of that content via XQuery Update.

In addition, TNTBase provides a plugin architecture for document format-specific customizations [35]. Using OMDoc as concrete syntax for MMT and the MMT API as a TNTBase plugin, we have made TNTBase MMT-aware so that data-intensive MMT algorithms can be executed within the database.

The TNTBase system and its documentation are available at <http://tntbase.org>. Below we describe some of the features particularly relevant for scalability.

4.1 Generating Content

Large scale collaborative authoring of mathematical documents requires **distributed** and **versioned** storage. On the language end, MMT supports this by making all identifiers URIs so that MMT documents can be distributed among authors and networks and reference each other. On the database end, TNTBase supports this by acting as a versioned MMT database.

In principle, versioning and distribution could also be realized with a plain SVN server. But for mathematics, it is important that the storage backend is aware of at least some aspects of the mathematical semantics. In large scale authoring processes, an important requirement is to guarantee consistency, i.e., it should be possible to reject commits of invalid documents. Therefore, TNTBase supports document format-specific **validation**.

For scalability, it is crucial that validation of interlinked collections of MMT documents is **incremental**, i.e., only those documents added or changed during a commit are validated. This is a significant effect because the committed documents almost always import modules from other documents that are already in the database, and these should not be revalidated — especially not if they contain unnecessary modules that introduce further dependencies.

Therefore, we integrate MMT separate compilation into TNTBase. During a commit TNTBase validates all committed files structurally by calling the MMT API. After successful validation, the ABox is generated and immediately stored in TNTBase. References to previously committed files are not resolved; instead their generated ABox is used for validation. Thus, validation is limited to the committed documents.

Case Study 7: In the LATIN project [18], we create an atlas of logics and logic translations formalized in MMT. At the current early stage of the project 5 people are actively editing so far about 100 files. These contain about 200 theories and 50 views, which form a single highly inter-linked MMT theory graph. We use TNTBase as the validity-guaranteeing backend storage.

The LATIN theory graph is highly modular. For example, the document giving the set-theoretical model theory of first-order logic from [16] depends on about 100 other theories. (We counted them conveniently using an XQuery, see below.) Standalone validation of this document takes about 15 seconds if needed files are retrieved from a local file system. Using separate compilation in TNTBase, it is almost instantaneous.

In fact, we can configure TNTBase so that structural validation is preceded by RelaxNG validation. This permits the MMT application to drop inefficient checks for syntax errors. Similarly, structural validation could be preceded by foundation-specific validation, but often we do not have a well-understood notion of separate compilation for specific foundations. But even in that case, we can do better than naive revalidation. MMT is designed so that it is foundation-independent which modules a given document depends on. Thus, we can collect these modules in one document using an efficient XQuery (see below) and then revalidate only this document. Moreover, we can use the presentation algorithm from Sect. 3.3 to transform the generated document into the input syntax of a dedicated implementation.

4.2 Retrieving Content

While the previous section already showed some of the strength of an MMT-aware TNTBase, its true strength lies in retrieving content. As every XML-native database, TNTBase supports XQuery but extends the DB XML syntax by a notion of file system path to address path-based collections of documents. Furthermore, it supports indexing to improve performance of queries and the querying of previous revisions. Finally, custom XQuery modules can be integrated into TNTBase.

MMT-aware retrieval is achieved through two measures. Firstly, **ABox caching** means that for every committed file, the MMT ABox is generated and stored in TNTBase. The ABox contains only two kinds of declarations — instances of unary and binary predicates — and is stored as a simple XML document. The element types in these documents are **indexed**, which yields efficient global queries.

Example 3. An MMT document for the algebra example from Sect. 2.1 is served at <http://alpha.tntbase.mathweb.org/repos/cds/math/algebra/algebra1.omdoc>. Its ABox is cached at <http://alpha.tntbase.mathweb.org:8080/tntbase/cds/restful/integration/validation/mmt/content/math/algebra/algebra1.omdoc>.

Secondly, custom **XQuery functions** utilize the cached and indexed ABoxes to provide efficient access to frequently needed aggregated documents. These include in particular the forward and backward dependency cones of a module. The backward dependency cone of a module M is the minimal set of modules needed to make M well-formed. Dually, the forward cone contains all modules that need M . If it were not for the indexed ABoxes, the latter would be highly expensive to compute: linear in the size of the database.

Case Study 8: The MMT presentation algorithm described in Sect. 3.3 takes a set of notations as input. However, additional notations may be given in imported theories, typically format-independent notations such as the one making `?Magma?* infix`. Therefore, when an MMT expression is rendered, all imported theories must be traversed for the sole reason of obtaining all notations.

Without MMT awareness in TNTBase, this would require multiple successive queries which is particularly harmful when presentation is executed locally while the imported theories are stored remotely. But even when all theories are available on a local disk, these successive calls already take 1.5 seconds for the above algebra document. (Once the notations are retrieved, the presentation itself is instantaneous.)

In MMT-aware TNTBase, we can retrieve all notations in the backward dependency closure of the presented expression with a single XQuery. ABox-indexing made this instantaneous up to network lag.

TNTBase does not only permit the efficient retrieval of such generated documents, but it also permits to commit edited versions of them. We call these **virtual documents** in [35]. These are essentially “XML database views” analogous to views in relational databases. They are editable, and TNTBase transparently patches the differences into the original files in the underlying versioning system.

Case Study 9: While manual refactoring of large theory graphs is as difficult as refactoring large software, there is virtually no tool support for it. For MMT, we obtain a simple renaming tool using a virtual document for the one-step (i.e., non-transitive) forward dependency cone of a theory T (see [35] for an example). That contains all references to T so that T can be renamed and all references modified in one go.

5 Conclusion and Future Work

This paper aims to pave the way for MKM “in the large” by proposing a theoretical and technological basis for a “Universal Digital Mathematics Library” (UDML) which has been touted as the grand challenge for MKM. In a nutshell, we conclude that the problem of scalability has to be addressed on all levels: we need modularity and accessibility of induced declarations in the representation format, incrementality and memoization in the implementation of the fundamental algorithms, and a mass storage solution that

supports fragment access and indexing. We have developed prototypical implementations and tested them on a variety of case studies.

The next step will be to integrate the parts and assemble a UDML installation with these. We plan to use the next generation of the OMDOC format, which will integrate the MMT infrastructure described in this paper as an interoperability layer; see [20] for a discussion of the issues involved. In the last years, we have developed OMDOC translation facilities for various fully formal theorem proving systems and their libraries. In the LATIN project [18], we are already developing a graph of concrete “logics-as-theories” to make the underlying logics interoperable.

References

1. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
2. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, Internet Engineering Task Force, 2005.
3. Y. Bertot and P. Castéran. *Coq’Art: The Calculus of Inductive Constructions*. Springer, 2004.
4. N. Bourbaki. *Theory of Sets*. Elements of Mathematics. Springer, 1968.
5. N. Bourbaki. *Algebra I*. Elements of Mathematics. Springer, 1974.
6. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
7. CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2960 of *LNCS*. Springer, 2004.
8. H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
9. S. Dumbrava, F. Horozal, and K. Sojakova. A Case Study on Formalizing Algebra in a Module System. In F. Rabe and C. Schürmann, editors, *Workshop on Modules and Libraries for Proof Assistants*, volume 429 of *ACM International Conference Proceeding Series*, pages 11–18, 2009.
10. W. Farmer. An Infrastructure for Intertheory Reasoning. In D. McAllester, editor, *Conference on Automated Deduction*, pages 115–131. Springer, 2000.
11. W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.
12. William M. Farmer. Mathematical Knowledge Management. In David G. Schwartz, editor, *Encyclopedia of Knowledge Management*, pages 599–604. Idea Group Reference, 2005.
13. J. Giceva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette and L. Dixon and C. Sacerdoti Coen and S. Watt, editor, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2009.
14. J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.
15. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. In *Fourth Workshop on Logical and Semantic Frameworks, with Applications*, volume 256 of *Electronic Notes in Theoretical Computer Science*, pages 49–65, 2009.
16. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. Under review, see http://kwarc.info/frabe/Research/EArabe_folsound_10.pdf, 2010.

17. W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
18. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. See <https://trac.omdoc.org/LATIN/>.
19. M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier and J. Campbell and J. Rubio and V. Sorge and M. Suzuki and F. Wiedijk, editor, *Mathematical Knowledge Management*, volume 5144 of *Lecture Notes in Computer Science*, pages 504–519, 2008.
20. M. Kohlhase, F. Rabe, and C. Sacerdoti Coen. A Foundational View on Integration Problems. Submitted to CALCULEMUS, 2010.
21. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
22. P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001.
23. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
24. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.
25. S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.
26. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
27. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
28. A. Poswolsky and C. Schürmann. System Description: Delphin A Functional Programming Language for Deductive Systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, pages 135–141. ENTCS, 2008.
29. F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. Available at <http://kwarc.info/frabe/Research/phdthesis.pdf>.
30. F. Rabe. The MMT System, 2008. See <https://trac.kwarc.info/MMT/>.
31. F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
32. D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.
33. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
34. V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3. Mulberry Technologies, Inc., 2009.
35. V. Zholudev, M. Kohlhase, and F. Rabe. A [insert XML Format] Database for [insert cool application]. In *Proceedings of XMLPrague*. XMPPrague.cz, 2010.